DSBA Codebook

greatlearning
Learning for Life

Preface

Data Science is the art and science of solving real world problems and making data driven decisions. It involves an amalgamation of three aspects and a good data scientist has expertise in all three of them. These are:

1) Mathematical/ Statistical understanding
2) Coding/ Technology understanding
3) Domain knowledge

Your lack of expertise should not become an impediment in your journey in Data Science. With consistent effort, you can become fairly proficient in coding skills over a period of time. This Codebook is intended to help you become comfortable with the finer nuances of Python and can be used as a handy reference for anything related to data science codes throughout the program journey and beyond that.

In this document we have followed the following syntax:

- Brief description of the topic
- Followed with a code example.

Please keep in mind there is no one right way to write a code to achieve an intended outcome. There can be multiple ways of doing things in Python. The examples presented in this document use just one of the approaches to perform the analysis. Please explore by yourself different ways to perform the same thing.

Contents

Table of Figures

Table of Equations

Supervised Learning – Regression

Linear Regression

Simple Linear Regression

$$y = mx + c$$

Equation 1:Simple Linear Regression Equation

Where Y is the dependent variable (target variable), X is the independent variable, m is the slope, and C is the Intercept

Multiple Linear Regression

$$y = m_1x_1 \; + \; m_2x_2 \; + \cdots . + m_nx_n \; + c$$

Equation 2:Multiple Linear Regression Equation:

Loss Function

$$L = \sum (Y_i - \widehat{Y_i})^2$$

Equation 3:Loss Function

More Info: sklearn

```
from sklearn.linear_model import LinearRegression
regression_model = LinearRegression()
regression_model.fit(x_train, y_train)
```

<u>Method 2:</u>

```
import statsmodels.api as sm
X = sm.add_constant(dataframe of predictor variables)
regression_model = sm.OLS(Dependent variable,X).fit()
```

To Predict:

```
Regression_model.predict(X)
```

<u>Method 3:</u>

import statsmodels.formula.api as SM
model = SM.ols(formula='Dependent Variable ~ $\sum_k Independent\ Variables_k$',data = 'Data Frame containing the required values').fit()
model.sumary() #this is to check the various parameters of the regression model

To Predict:
model.predict('Dataframe containing the predictor variables')

Bias-Variance Trade-Off in Multiple Regression

Both the bias and the variance are desired to be low, as large values of both can result in poor predictions from the model

- Bias: The model is not learning anything (under-fitting)
- Variance: The model is over-fitting and train score is very high (~1.0) while test score is very low.

The trade-off is a point of moderate Bias and Variance to make a model with good prediction power

The Linear Regression can have a high variance because of the following:

- The predictors are highly correlated with each other;
- There are too many predictors

The general solution to this is to reduce variance by introducing some bias. This approach is called regularization

LASSO Regression (Least Absolute Shrinkage and Selection Operator)

L1 Regularization penalizes coefficients with the sum of their absolute values. For high values of λ, many coefficients can shrink to 0

The cost function is defined as:

$$L = \sum(Y_i - \widehat{Y_i})^2 + \lambda\,|\beta|$$

Equation 4:Loss Function for LASSO

As λ approaches 0, LASSO Regression Coefficients tends to become same as Linear Regression Coefficients

Lasso can set some coefficients to zero, thus performing variable selection

```
from sklearn.linear_model import Lasso
lasso = Lasso(alpha=200)
lasso.fit(x_train, y_train)
```

More Info: [sklearn]

Ridge Regression

L2 Regularization

The cost function is used to minimize the SSE but also penalize the coefficients so as to shrink them towards 0, however none of the coefficient will become zero (This is where Ridge Regression differs from LASSO)

$$L = \sum(Y_i - \widehat{Y_i})^2 + \lambda\,\beta^2$$

Equation 5:Loss Function for Ridge

As λ approaches 0, Ridge Regression Coefficients tends to become same as Linear Regression Coefficients

- Scaling is necessary for ridge regression for giving equal penalty to coefficients and it also lowers λ values

```
from sklearn.linear_model import Ridge
ridge = Ridge()
ridge.fit(x_train_scaled, y_train)# scale the data for Ridge Regression
```

More Info: [sklearn]

Elastic Net

Linear regression SSE with combination of Ridge and LASSO:

$$L = \sum(Y_i - \widehat{Y_i})^2 + \lambda_1 |\beta| + \lambda_2 \ \beta^2$$

Equation 6:Loss Function for Elastic Net

```
from sklearn.linear_model import ElasticNet
eln = ElasticNet()
eln.fit(x_train, y_train) #or
eln.fit(x_train_scaled, y_train) #or scale the data
```

More Info: sklearn

Model Evaluation – Regression

Root Mean Squared Error

```
from sklearn.metrics import mean_squared_error
import numpy as np
print("Train RMSE score:",np.sqrt(mean_squared_error(y_train,regression_model.predict(x_train))))
```

R Square

```
from sklearn.metrics import r2_score
print("Train R Squared:", r2_score(y_train,regression_model.predict(x_train)))
```

Supervised Learning – Classification

Naive Bayes

Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of conditional independence between every pair of features given the value of the class variable. Bayes' theorem states the following relationship, given class variable y and dependent feature vector x_1 through x_n

$$P(y|x_1, \dots, x_n) = \frac{P(y)P(\, x_1, \dots, x_n|y)}{P(x_1, \dots, x_n)}$$

Naive Bayes learners and classifiers can be extremely fast compared to more sophisticated methods. The decoupling of the class conditional feature distributions means that each distribution can be independently estimated as a one dimensional distribution. This in turn helps to alleviate problems stemming from the curse of dimensionality.

GaussianNB implements the Gaussian Naive Bayes algorithm for classification

source: scikit-learn

```
from sklearn.naive_bayes import GaussianNB

# Invoking the NB Gaussian function to create the model

# fitting the model in the training data set
```

```
model = GaussianNB()

model.fit(x_train, y_train)

y_predict = model.predict(x_test)

model_score = model.score(x_test, y_test)

print(model_score)

print(metrics.confusion_matrix(y_test, y_predict))
```

Logistic Regression

Logistic regression is a linear model for classification rather than regression. It is also known as logit regression. In this model, the probabilities describing the possible outcomes of a single trial are modelled using a logistic function.

source: scikit-learn

```
from sklearn.linear_model import LogisticRegression

model = LogisticRegression(max_iter=1000)

model.fit(x_train, y_train)

y_predict = model.predict(x_test)

model_score = model.score(x_test, y_test)

print(model_score)

print(metrics.confusion_matrix(y_test, y_predict))
```

Method 2:

```
import statsmodels.formula.api as SM
model = SM.ols(formula='Dependent Variable ~ $\sum_k Independent\ Variables_k$',data = 'Data Frame containing the required values').fit()
model.summary()#to check the parameters
model.predict('Data Frame containing the predictor variabes') #in this step we only get the probability values
#For the class prediction, we need to pass the following lines of code

y_pred_class=[]
for i in range(0,len(y_train_prob)):

    if np.array(y_train_prob)[i]>0.5: #numpy has been imported under the alias np

        a=1

    else:

        a=0

    y_pred_class.append(a)
```

The object y_pred_class has the required predicted classes based on a cut-off of 0.5. If a particular probability value is greater than 0.5, we have marked that particular class as 1 and if the value is less than 0.5 then that particular class has been marked as 0.

Note: The default parameters in the sklearn's logistic regression function imposes a default regularization of L2 (Ridge).

Decision Tree

Decision Trees (DTs) are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features.

Deeper the tree, more complex are the decision rules and model could be over fitted

source: scikit-learn

```
#Visualising Dtree in Python

from sklearn.datasets import load_iris

from sklearn import tree

import matplotlib.pyplot as plt

clf = tree.DecisionTreeClassifier()

iris=load_iris()

plt.figure(figsize=(12,7))

tree.plot_tree(clf.fit(iris.data, iris.target),filled=True)

plt.show()
```

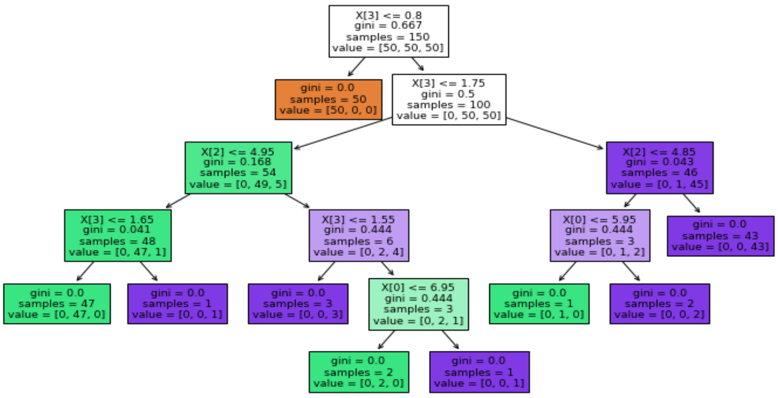

Figure 1:Decision Tree

```
from sklearn import tree

clf = tree.DecisionTreeClassifier()

clf = clf.fit(x_train, y_train)

y_predict = clf.predict(x_test)

model_score = clf.score(x_test, y_test)

print(model_score)

print(metrics.confusion_matrix(y_test, y_predict))
```

Ensemble: Random Forest

In random forests, each tree in the ensemble is built from a sample drawn with replacement (i.e., a bootstrap sample) from the training set

Furthermore, when splitting each node during the construction of a tree, the best split is found either from all input features or a random subset of size max_features

The purpose of these two sources of randomness is to decrease the variance of the forest estimator. Indeed, individual decision trees typically exhibit high variance and tend to overfit. The injected randomness in forests yield decision trees with somewhat decoupled prediction errors. By taking an average of those predictions, some errors can cancel out. Random forests achieve a reduced variance by combining diverse trees, sometimes at the cost of a slight increase in bias. In practice the variance reduction is often significant hence yielding an overall better model

The scikit-learn implementation combines classifiers by averaging their probabilistic prediction

source: scikit-learn

```
#Import Random Forest Model

from sklearn.ensemble import RandomForestClassifier

#Create a Gaussian Classifier

clfRF=RandomForestClassifier(n_estimators=100)

#Train the model using the training sets y_pred=clf.predict(X_test)

clfRF.fit(x_train,y_train)

y_pred=clfRF.predict(x_test)

model_scoreRF = clfRF.score(x_test, y_test)
```

Support Vector Machine

Support vector machines (SVMs) are a set of supervised learning methods used for classification, regression and outliers' detection

source: scikit-learn

```
from sklearn import svm


clfSVM = svm.SVC()

clfSVM.fit(x_train, y_train)

y_pred=clfSVM.predict(x_test)

model_scoreSVM = clfSVM.score(x_test, y_test)

print(model_scoreSVM)

print(metrics.confusion_matrix(y_test, y_pred))
```

Linear Discriminant Analysis

LDA can be derived from simple probabilistic models, which model the class conditional distribution of the data for each class .Predictions can then be obtained by using Bayes’ rule:

and we select the class which maximizes this conditional probability.

More specifically, for linear discriminant analysis, $P(X|y)$P(X|y) is modeled as a multivariate Gaussian distribution with density:

where dd is the number of features.

source: scikit-learn

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

clfLDA = LinearDiscriminantAnalysis()

clfLDA.fit(x_train, y_train)

y_pred=clfLDA.predict(x_test)

model_scoreLDA = clfLDA.score(x_test, y_test)

print(model_scoreLDA)

print(metrics.confusion_matrix(y_test, y_pred))
```

Ensemble: Bagging

Bagging classifier

A Bagging classifier is an ensemble meta-estimator that fits base classifiers each on random subsets of the original dataset and then aggregate their individual predictions (either by voting or by averaging) to form a final prediction. Such a meta-estimator can typically be used as a way to reduce the variance of a black-box estimator (e.g., a decision tree), by introducing randomization into its construction procedure and then making an ensemble out of it.

source: scikit-learn

```
from sklearn.svm import SVC

from sklearn.ensemble import BaggingClassifier

from sklearn.datasets import make_classification

X, y = make_classification(n_samples=100, n_features=4, n_informative=2, n_redundant=0,random_state=0, shuffle=False)

clf = BaggingClassifier(base_estimator=SVC(), n_estimators=10, random_state=0).fit(X, y)

clf.predict([[0, 0, 0, 0]])
```

Ensemble: Boosting

The module sklearn.ensemble includes the popular boosting algorithm AdaBoost, introduced in 1995 by Freund and Schapire

The core principle of AdaBoost is to fit a sequence of weak learners (i.e., models that are only slightly better than random guessing, such as small decision trees) on repeatedly modified versions of the data. The predictions from all of them are then combined through a weighted majority vote (or sum) to produce the final prediction.

The number of weak learners is controlled by the parameter n_estimators. The learning_rate parameter controls the contribution of the weak learners in the final combination. By default, weak learners are decision stumps. Different weak learners can be specified through the base_estimator parameter. The main parameters to tune to obtain good results are n_estimators and the complexity of the base estimators (e.g., its depth max_depth or minimum required number of samples to consider a split min_samples_split).

source: scikit-learn

```
from sklearn.ensemble import AdaBoostClassifier
clfADB = AdaBoostClassifier(n_estimators=100)
clfADB.fit(x_train,y_train)
y_pred=clfADB.predict(x_test)
model_scoreADB = clfADB.score(x_test, y_test)
print(model_scoreADB)
print(metrics.confusion_matrix(y_test, y_pred))
```

Xtreme Gradient Boosting

XGBoost stands for “Extreme Gradient Boosting”, where the term “Gradient Boosting” originates from the paper Greedy Function Approximation: A Gradient Boosting Machine, by Friedman.

XGBoost algorithm was developed as a research project at the University of Washington. Tianqi Chen and Carlos Guestrin presented their paper at SIGKDD Conference in 2016 and caught the Machine Learning world by fire

XGBoost is an optimized distributed gradient boosting library designed to be highly efficient, flexible and portable. It implements machine learning algorithms under the Gradient Boosting framework. XGBoost provides a parallel tree boosting (also known as GBDT, GBM) that solve many data science problems in a fast and accurate way

source: xgboost

```
import xgboost as xgb
clfXGB=xgb.XGBClassifier(random_state=1,learning_rate=0.01)
clfXGB.fit(x_train, y_train)
y_pred=clfXGB.predict(x_test)
model_scoreXGB=clfXGB.score(x_test,y_test)
print(model_scoreXGB)
print(metrics.confusion_matrix(y_test,y_pred))
```

K-Nearest Neighbors

Neighbors-based classification is a type of instance-based learning or non-generalizing learning: it does not attempt to construct a general internal model, but simply stores instances of the training data. Classification is computed from a simple majority vote of the nearest neighbors of each point: a query point is assigned the data class which has the most representatives within the nearest neighbors of the point.

source: scikit-learn

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler

ss=StandardScaler()

x_train_scaled=ss.fit_transform(x_train) #scaling the data since KNN is a distance based algorithm.

x_test_scaled=ss.transform(x_test)

clfKNN=KNeighborsClassifier()

clfKNN.fit(x_train_scaled,y_train)

y_pred=clfKNN.predict(x_test_scaled)

model_scoreKNN = clfKNN.score(x_test_scaled, y_test)

print(model_scoreKNN)

print(metrics.confusion_matrix(y_test, y_pred))
```

Model Tuning

Cross-validation

Learning the parameters of a prediction function and testing it on the same data is a methodological mistake: a model that would just repeat the labels of the samples that it has just seen would have a perfect score but would fail to predict anything useful on yet-unseen data. This situation is called overfitting. To avoid it, it is common practice when performing a (supervised) machine learning experiment to hold out part of the available data as a test set X_test, y_test. Here is a flowchart of typical cross validation workflow in model training. The best parameters can be determined by grid search techniques

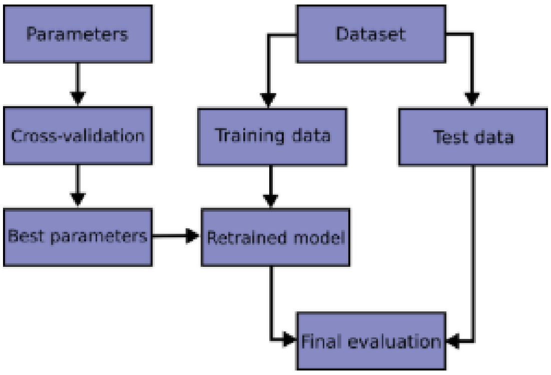

Figure 2:Grid Search

When evaluating different settings ("hyperparameters") for estimators, such as the C setting that must be manually set for an SVM, there is still a risk of overfitting on the test set because the parameters can be tweaked until the estimator performs optimally. This way, knowledge about the test set can "leak" into the model and evaluation metrics no longer report on generalization performance. To solve this problem, yet another part of the dataset can be held out as a so-called "validation set": training proceeds on the training set, after which evaluation is done on the validation set, and when the experiment seems to be successful, final evaluation can be done on the test set.

However, by partitioning the available data into three sets, we drastically reduce the number of samples which can be used for learning the model, and the results can depend on a particular random choice for the pair of (train, validation) sets.

source: scikit-learn

A solution to this problem is a procedure called cross-validation (CV for short). A test set should still be held out for final evaluation, but the validation set is no longer needed when doing CV. In the basic approach, called k-fold CV, the training set is split into k smaller sets (other approaches are described below, but generally follow the same principles). The following procedure is followed for each of the k “folds”:

A model is trained using k-1 folds as training data;

the resulting model is validated on the remaining part of the data (i.e., it is used as a test set to compute a performance measure such as accuracy)

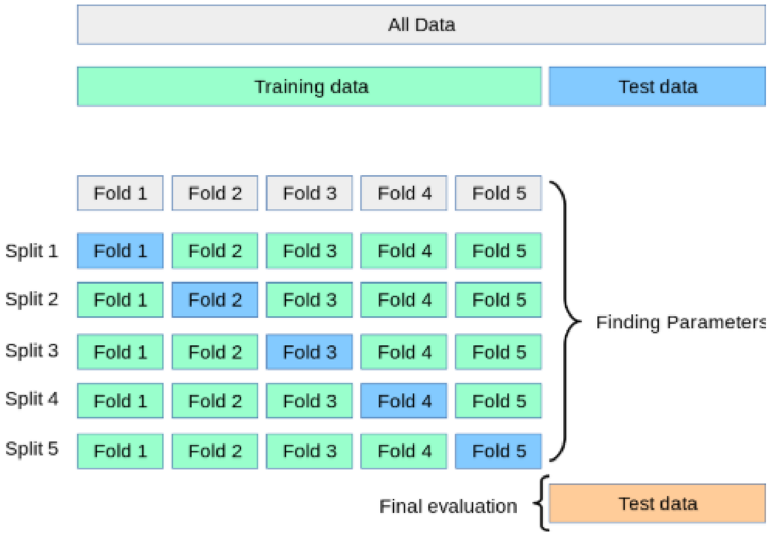

Figure 3: Cross Validation

```
from sklearn.model_selection import cross_val_score

clfCVRF = RandomForestClassifier(n_estimators=100)

scores = cross_val_score(clfCVRF, x_train, y_train, cv=10)

np.mean(scores)
```

Grid Search

It is an exhaustive search over specified parameter values for an estimator.

source: scikit-learn

```
from sklearn.model_selection import GridSearchCV

param_grid = {'max_depth': [7,10],'max_features': [4, 6],'min_samples_leaf': [3, 15,30],'min_samples_split': [30, 50,100],

    'n_estimators': [300, 500]}

rfr = RandomForestRegressor()

grid_search = GridSearchCV(estimator = rfr, param_grid = param_grid, cv = 3)

grid_search.fit(x_train,y_train)

print(grid_search.best_params_)
```